ADSA MiniProblem Report

Amir Mahmoudi
Yassine Lahbabi

December 2020

Contents

1 Introduction

Welcome to our Advanced Data Structure and Algorithm Project. This project needed to be done in a group of 2. We've done all this project in python, the purpose of this report is to explain our methodology, choices and discuss our results. Have a nice reading !

2 Main

For our main, we used a simple menu that works with the input of the user. Our main focus was that this main needed to be always shown to the user of our program so that he always have a back option if he wants to explore another function of the program. And a handle exception or error where the program tells you that you entered an invalid option so that the use of the program is as simple as it can be.
We also added a function to clear the console because some of our functions fill up the whole console window so it does not look aesthetic to the user.
Finally we added a sub-menu for the step 1 of our program so that the user have the choice between the different simulations that he wants to see.

3 Step 1 : Organize the tournament

3.1 Data Structure

3.1.1 Player

For the player we've use a class data structure.A player is defined by is name/id and his score (the score is initiated at None) hence when we create a player if we only specify is name, the score will be None. The player represent a node in our score data structure, you will see it in next section.
The other information regarding a player are :

- His score, if it's not mention when the player is created with attribute the value zero in the other case we keep the value. It's an int value.
- His life status, it's a Boolean initiated at True meaning he's alive.
- His role, the variable is self.impostor, it's a Boolean initiated at False meaning he's a crew mate.When the game starts this variable can change to True implies he's now an impostor.
- His children, the player is a node in our score data structure so we mention if he has a left or right child they are both Boolean initiated at False because the Score data structure is not created.

Other than the init function we have the str function return a string to describe the player for instance we can have the string :

Amir is alive, and he is an Impostor

3.1.2 Score

To store the score of all our players and have a leader board we decided to use an AVL Tree for various reason, first, the insertion and deletion was simple to implement we have also the possibility to re balance our AVL Tree. But why re balance is essential to this structure ? You will see this part in the In Order Traversal paragraph. At first we were going to use a list with hash-tables to store the players and then re-inject them to the different lists so that we can compare between them. But we figured out that the AVL Tree was the most optimized data structure to use as it compares between all the nodes in each side so that we can have a balanced tree at the end and a leader board with the players and their respective scores.

Insertion
For insertion, if the AVL tree is empty we initiate the player inserted as the root. If the AVL isn't empty we will insert the player node by comparing his score to the AVL's nodes. After inserting the player we re-balance our tree to keep it balanced.

Deletion
For deletion, we will use the player name as our reference key to delete it. First we check if we find the key in the leaf nodes of our tree, if we find him we delete the player. We also have to take in count the fact that the node of our tree could have one sub-tree, for this case we replace the root so that our self node is defined as a right or left node depending on if we are in the case of a left sub-tree or a right one.
If we do not find the player key defined, we check the smallest node in the right sub-tree or the predecessor as largest node in the left one.
After doing theses steps we can finally compare the player's score to the node score that we put before and we delete the player. The final step is to re-balance our tree after deleting the players without changing the scores so that our AVL Tree stays balanced.

Re-Balance
The Re-balance part is really important as we use it in every function that forces a change in our data structure. As we have seen before in our ADSA course, every time we insert or delete a node we have to check if our node is balanced or not.
This step is necessary because if it is not balanced we have to update the AVL's structure so that it remains balanced which is one of the most important condition of the AVL Tree.
Let's first break down our 2 main sub-functions that we use in our Re-balance main function :

- Updating the heights : A simple sub-function to keep track of the tree's height. A tree's height is the max height of either the left or right sub-trees + 1 as we have the root of the tree.
- Updating balances : Calculating the tree balance factor using the operation that we have seen before : factor = height of the left sub-tree - height of the right sub-tree.

Now that we have seen theses sub-functions let's focus on our main function, so first of all we check if we need to re-balance the tree, for each node that we have checked if our balance factor calculated remains -1,0 or +1 then no rotations are necessary and then our tree is perfectly balanced. If the left sub-tree is larger that right sub-tree or the other case (right larger that the left one) we rotate it so that our sub-trees are balanced. We use the Right or left rotation depending on the unbalanced case that we are in.

In Order Traversal
For In Order Traversal, we use a recursive method we are going through all the tree adding to a list, the left sub tree of a node, the node in question and the last the right sub tree of this node. Let's manually compute the function with this example.

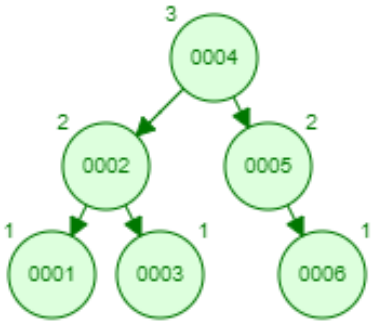

Figure 1: AVL Tree

First the algorithm will check the node 0004 but it has children so we check first his left child that's node 0002. This node has children so we check his left child that's node 0001. This node doesn't have children so we add it to our list, we add then the node 0002 we check is right child 0003 same reason for node 0001. We came back to node 0004 we add it to the list.
So far our list is [1,2,3,4]. We now at node 0004 and we need to check his right child, so we are now at node 0005, this node doesn't have a left child so we append him to our list and then check his right child. We're finally at node 0006, we've checked all the nodes and our list is : [1,2,3,4,5,6].
To conclude, the In Order Traversal is useful to obtain directly the leaderboard.

3.1.3 Tournament

As we have established before, we've used an AVL Tree as our main data structure and an in order traversal of our tree to have the updated leaderboard.
Now, knowing that we had to display the last 10 players and the issue of every game so that we can track back the 100 initial players and the last 10 we've decided to compute them all in one class "Tournament" where we display the rounds in which we have all the 100 initial players and we update the leaderboard at the end of each single game in the tournament as a set of 3 games so that we have our 10 last players.
We then proceed to the finals where we redo a set of games but this time to have the last 3 players that have won the tournament through a ranking displaying their scores.

3.2 Methodology

There are 2 main ways to simulate the tournament and we can subdivide these ways into :

3.2.1 Simulation of the Tournament

For the first way, we are doing a simple simulation of the tournament using functions to update the ladder and to have the 10 last players as seen before using the AVL Tree.
So we simply display the 10 rounds of 3 game for each set of player so that we can have a clear idea of what the issue of each game is and who won it.

3.2.2 Simulation of a game

For the second way, we are not displaying the whole tournament, but just one game. To see what a simulation of a game could be knowing that for each action or task a player gets an accurate number of points.
For that, we use a method to test the points attribution between players in a game.

4 Step 2 : You>Guybrush Threepwood>Professor Layton

To represent the relation "have seen" between players as a graph, we used a dictionary where the keys are the id's player (0,1,2,3,etc...) and the values are the player seen by the player in key so for instance the relation for player 0 is 0 : [1,4,5].
During the practice work session our professor said we should use graph coloring for this problem but in our opinion this method is not really helpful here why :

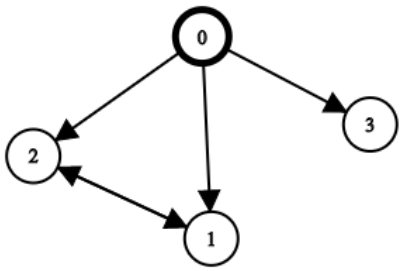

Figure 2: graph example

So for this graph we have :

- 0 has been killed
- 0 has seen 1,2 and 3
- 1 and 2 have seen each other

From those assumptions we can say that 1,2 and 3 are suspects if we use graph coloring we can say that 0 is blue, 1 is yellow, 2 is red and 3 is yellow. For our pairs of impostors we only have 3 with 1 and 3 with 2 because 1 and 2 have seen each other so they can't be both impostors. The problem here is that in one case the impostors have the same color (yellow) and in the other case they have different colors.
We can then conclude that the color can't afford a real argument in assuming which one is the impostor.
To answer this problem we have decided to use the dictionary and apply the rules mentioned in the exercise. So first our 3 main suspect are 1, 4 and 5 because they have been seen by player 0 so we need to find each pair for 1, 4 and 5. From the rules we know that impostors can't see each other then the potential pair for each suspect are all the players that the suspect hasn't seen.
Our output is :

```
{0: 'Red', 1: 'Blue', 2: 'Red', 3: 'Blue', 4: 'Green', 5: 'Blue', 6: 'Red', 7: 'Green',
 8: 'Green', 9: 'Blue'}
For the suspect player 1 we have his potential partner :
[3, 4, 5, 7, 8, 9]
For the suspect player 4 we have his potential partner :
[1, 2, 5, 6, 7, 8]
For the suspect player 5 we have his potential partner :
[1, 2, 3, 4, 6, 9]
```

Figure 3: Step 2 output

5 Step 3 : I don't see him, but I can give proofs he vents !

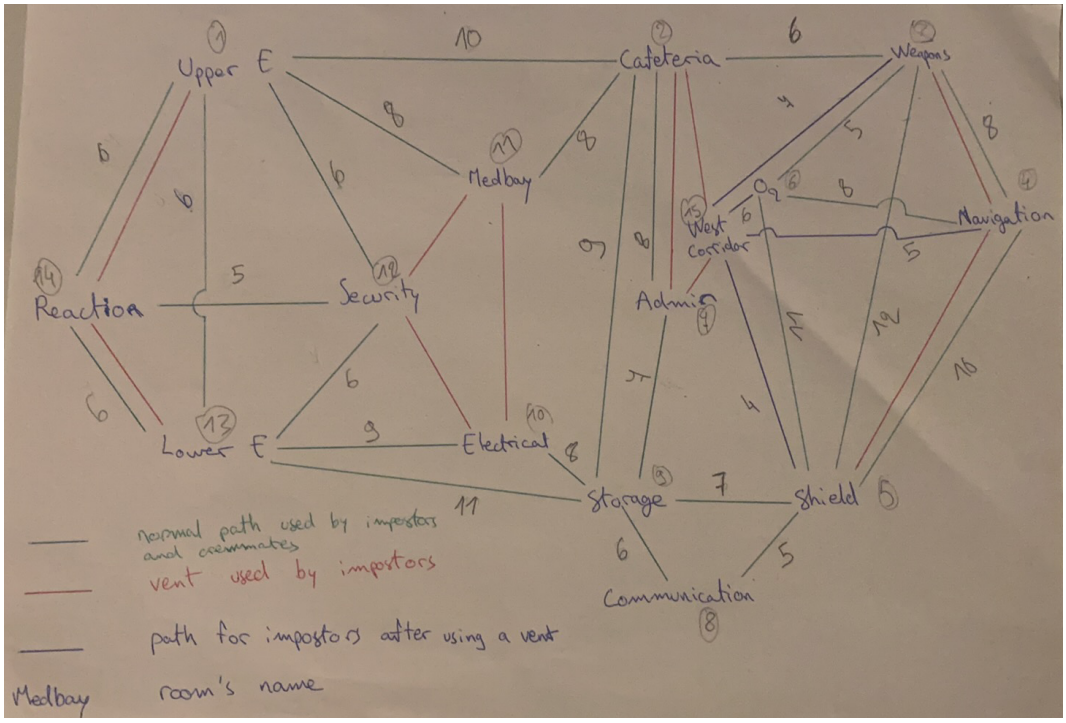

Figure 4: Weighted Graph

We need two models for the graph because the impostors can use an extra vertex : West Corridor and also we need to change the weight value, for instance going from Upper E to Reactor will take 6 seconds for a crew mate but an impostor can use the vent so we have a weight of zero.Hence, we need to implement a graph for the impostors and another for the crew mates.

We obtained the weights in figure right above by measuring with a ruler the distance between each rooms. Each centimeter is a weight of 1
First, we used Dijkstra Algorithm but the inconvenient was that we compute only for one vertex so if we wanted all pair of vertices the time complexity will be too big. So we have chosen to use the Floyd Warshall algorithm because it computes all path between all pair of vertices.
By running the code we have the time travel between a pair of rooms.

	Upper E	Cafeteria	Weapons	Navigations	Shield	O2	Admin	Communication	Storage	Electrical	MedBay	Security	Lower E	Reactor	West Corridor
Upper E	0.0	10.0	14.0	14.0	14.0	16.0	10.0	17.0	11.0	5.0	5.0	5.0	0.0	0.0	10.0
Cafeteria	10.0	0.0	4.0	4.0	4.0	6.0	0.0	9.0	7.0	8.0	8.0	8.0	10.0	10.0	0.0
Weapons	14.0	4.0	0.0	0.0	0.0	5.0	4.0	5.0	7.0	12.0	12.0	12.0	14.0	14.0	4.0
Navigations	14.0	4.0	0.0	0.0	0.0	5.0	4.0	5.0	7.0	12.0	12.0	12.0	14.0	14.0	4.0
Shield	14.0	4.0	0.0	0.0	0.0	5.0	4.0	5.0	7.0	12.0	12.0	12.0	14.0	14.0	4.0
O2	16.0	6.0	5.0	5.0	5.0	0.0	6.0	10.0	12.0	14.0	14.0	14.0	16.0	16.0	6.0
Admin	10.0	0.0	4.0	4.0	4.0	6.0	0.0	9.0	7.0	8.0	8.0	8.0	10.0	10.0	0.0
Communication	17.0	9.0	5.0	5.0	5.0	10.0	9.0	0.0	6.0	14.0	14.0	14.0	17.0	17.0	9.0
Storage	11.0	7.0	7.0	7.0	7.0	12.0	7.0	6.0	0.0	8.0	8.0	8.0	11.0	11.0	7.0
Electrical	5.0	8.0	12.0	12.0	12.0	14.0	8.0	14.0	8.0	0.0	0.0	0.0	5.0	5.0	8.0
MedBay	5.0	8.0	12.0	12.0	12.0	14.0	8.0	14.0	8.0	0.0	0.0	0.0	5.0	5.0	8.0
Security	5.0	8.0	12.0	12.0	12.0	14.0	8.0	14.0	8.0	0.0	0.0	0.0	5.0	5.0	8.0
Lower E	0.0	10.0	14.0	14.0	14.0	16.0	10.0	17.0	11.0	5.0	5.0	5.0	0.0	0.0	10.0
Reactor	0.0	10.0	14.0	14.0	14.0	16.0	10.0	17.0	11.0	5.0	5.0	5.0	0.0	0.0	10.0
West Corridor	10.0	0.0	4.0	4.0	4.0	6.0	0.0	9.0	7.0	8.0	8.0	8.0	10.0	10.0	0.0

Figure 5: Floyd Warshall Matrix for Impostors

	Upper E	Cafeteria	Weapons	Navigations	Shield	O2	Admin	Communication	Storage	Electrical	MedBay	Security	Lower E	Reactor
Upper E	0.0	10.0	16.0	24.0	24.0	21.0	18.0	23.0	17.0	15.0	8.0	6.0	6.0	6.0
Cafeteria	10.0	0.0	6.0	14.0	16.0	11.0	8.0	15.0	9.0	17.0	8.0	16.0	16.0	16.0
Weapons	16.0	6.0	0.0	8.0	12.0	5.0	14.0	17.0	15.0	23.0	14.0	22.0	22.0	22.0
Navigations	24.0	14.0	8.0	0.0	10.0	8.0	22.0	15.0	17.0	25.0	22.0	30.0	28.0	30.0
Shield	24.0	16.0	12.0	10.0	0.0	11.0	14.0	5.0	7.0	15.0	24.0	24.0	18.0	24.0
O2	21.0	11.0	5.0	8.0	11.0	0.0	19.0	16.0	18.0	26.0	19.0	27.0	27.0	27.0
Admin	18.0	8.0	14.0	22.0	14.0	19.0	0.0	13.0	7.0	15.0	16.0	24.0	18.0	24.0
Communication	23.0	15.0	17.0	15.0	5.0	16.0	13.0	0.0	6.0	14.0	23.0	23.0	17.0	23.0
Storage	17.0	9.0	15.0	17.0	7.0	18.0	7.0	6.0	0.0	8.0	17.0	17.0	11.0	17.0
Electrical	15.0	17.0	23.0	25.0	15.0	26.0	15.0	14.0	8.0	0.0	23.0	15.0	9.0	15.0
MedBay	8.0	8.0	14.0	22.0	24.0	19.0	16.0	23.0	17.0	23.0	0.0	14.0	14.0	14.0
Security	6.0	16.0	22.0	30.0	24.0	27.0	24.0	23.0	17.0	15.0	14.0	0.0	6.0	5.0
Lower E	6.0	16.0	22.0	28.0	18.0	27.0	18.0	17.0	11.0	9.0	14.0	6.0	0.0	6.0
Reactor	6.0	16.0	22.0	30.0	24.0	27.0	24.0	23.0	17.0	15.0	14.0	5.0	6.0	0.0

Figure 6: Floyd Warshall Matrix for Crew mates

```
impostors going from Lower E to Upper E is 6.0 seconds quicker than a crewmate
impostors going from Lower E to Cafeteria is 6.0 seconds quicker than a crewmate
impostors going from Lower E to Weapons is 8.0 seconds quicker than a crewmate
impostors going from Lower E to Navigations is 14.0 seconds quicker than a crewmate
impostors going from Lower E to Shield is 4.0 seconds quicker than a crewmate
impostors going from Lower E to O2 is 11.0 seconds quicker than a crewmate
impostors going from Lower E to Admin is 8.0 seconds quicker than a crewmate
impostors going from Lower E to Communication is 0.0 seconds quicker than a crewmate
impostors going from Lower E to Storage is 0.0 seconds quicker than a crewmate
impostors going from Lower E to Electrical is 4.0 seconds quicker than a crewmate
impostors going from Lower E to MedBay is 9.0 seconds quicker than a crewmate
impostors going from Lower E to Security is 1.0 seconds quicker than a crewmate
impostors going from Lower E to Reactor is 6.0 seconds quicker than a crewmate
```

Figure 7: Time travel comparison

To obtain the comparison between those matrix we just substract the impostor matrix from the crew mate matrix without paying attention to the west corridor values because Crew mates can't access to those paths. Hence we have 2 matrix of dimension 14x14. We also don't compute the comparison for travel time when the start and end room are the same prevent from useless output.

6 Step 4 : Secure the last task

For this step we represented the graph as a dictionary : the keys are the room's name and the values are the room's next to the key room. In order to find a route passing through each room only one time, we needed an algorithm with backtracking hence we used the Hamilton algorithm. This method is recursive we refer first the starting room and the final room.The algorithm will compute for each room in the dictionary.
When we implement the code we have this :

```
Starting room : Communication ---> Ending room : Medbay

The paths passing by all rooms are :

['Communication', 'Shield', 'Navigation', 'O2', 'Weapons', 'Cafeteria', 'Admin', 'Storage', '
Electrical', 'Lower E', 'Security', 'Reactor', 'Upper E', 'Medbay']

['Communication', 'Shield', 'Navigation', 'O2', 'Weapons', 'Cafeteria', 'Admin', 'Storage', '
Electrical', 'Lower E', 'Reactor', 'Security', 'Upper E', 'Medbay']

['Communication', 'Shield', 'O2', 'Navigation', 'Weapons', 'Cafeteria', 'Admin', 'Storage', '
Electrical', 'Lower E', 'Security', 'Reactor', 'Upper E', 'Medbay']

['Communication', 'Shield', 'O2', 'Navigation', 'Weapons', 'Cafeteria', 'Admin', 'Storage', '
Electrical', 'Lower E', 'Reactor', 'Security', 'Upper E', 'Medbay']



Starting room : Communication ---> Ending room : Security

Sorry, we don't have path passing by all the rooms

Starting room : Communication ---> Ending room : Lower E

Sorry, we don't have path passing by all the rooms
```

Figure 8: Hamilton output

We've made another function,Hamilton algorithm finding the quickest path for a pair of rooms.It works the same way as the other Hamilton algorithm but we only keep the path with the fewest rooms.

```
Starting room : Security ---> Ending room : Upper E

The quickest path found is : ['Security', 'Upper E']

Starting room : Security ---> Ending room : Cafeteria

The quickest path found is : ['Security', 'Upper E', 'Cafeteria']

Starting room : Lower E ---> Ending room : Upper E

The quickest path found is : ['Lower E', 'Storage', 'Cafeteria', 'Upper E']

Starting room : Lower E ---> Ending room : Cafeteria

The quickest path found is : ['Lower E', 'Storage', 'Cafeteria']

Starting room : Lower E ---> Ending room : Storage

The quickest path found is : ['Lower E', 'Storage']

Starting room : Reactor ---> Ending room : Upper E

The quickest path found is : ['Reactor', 'Upper E']

Starting room : Reactor ---> Ending room : Cafeteria

The quickest path found is : ['Reactor', 'Upper E', 'Cafeteria']
```

Figure 9: Hamilton output